CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Kent Academic Repository

Is the Quality of Numerical Subroutine Code Improving?

T. R. Hopkins[1]

ABSTRACT We begin by using a software metric tool to generate a number of software complexity measures and we investigate how these values may be used to determine subroutines which are likely to be of substandard quality.
Following this we look at how these metric values have changed over the years. First we consider a number of freely available Fortran libraries (Eispack, Linpack and Lapack) which have been constructed by teams. In order to ensure a fair comparison we use a restructuring tool to transform original Fortran 66 code into Fortran 77.
We then consider the Fortran codes from the Collected Algorithms from the ACM (CALGO) to see whether we can detect the same trends in software written by the general numerical community.
Our measurements show that although the standard of code in the freely available libraries does appear to have improved over time these libraries still contain routines which are effectively unmaintainable and untestable. Applied to the CALGO codes the metrics indicate a very conservative approach to software engineering and there is no evidence of improvement, during the last twenty years, in the qualities under discussion.

1 Introduction

The last two decades have witnessed dramatic advances in the way we view the creation of software. Since *software engineering* was born in 1969 [BR70] an ever growing number of techniques have been put forward in an attempt to promote software production from an art to a science. For almost as long attempts have been made to quantify the quality of software. This has led to a bewildering number of software metrics being proposed; an excellent review of many of these may be found in the book by Zuse [Zus91]. In addition, a number of software tools have appeared which compute a variety

[1]Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.
E-mail: trh@ukc.ac.uk

This paper appears in *Modern Software Tools for Scientific Computing*, ed. Arge, Bruaset, and Langtangen, Birkhauser Boston, 1997, ISBN 0-8176-3974-8. Reprinted with permission.

of metrics for code written in most of the commonly used programming languages. Such tools include QAFortran [Pro92], QAC and QAC++, the nag_metrics tool from the NagWare f77 tool suite [Num92] and the LDRA Testbed [LDR].

In this paper we use a small number of these metrics to investigate whether we can detect an improvement in the quality of numerical software written in Fortran over the last twenty years. In section 2 we briefly discuss these metrics and, in the following section, we look at how successful they are at identifying software modules that suffer from readability, maintainability and testability problems.

We use the QAFortran tool to generate these metrics for the routines in a number of public domain packages. We also note, in this section, that during the time spanned by these libraries, the Fortran language has evolved from Fortran 66 [ANS66], through Fortran 77 [ANS79] to Fortran 90 [ISO91]. The additional control structures made available with each new code standard means that it is unfair to make direct comparisons of the code metrics between packages without taking into account the version of Fortran being used. In order to 'normalize' the metrics we have only considered code written in Fortran 66 and Fortran 77 and we have used an automatic code restructurer, *spag* [Pol93], to translate from Fortran 66 to Fortran 77. Metrics from the restructured Fortran 66 codes may then be compared.

The packages used in section 3 have all been produced by teams of researchers which have included wide ranges of expertise from software engineers to numerical analysts. To try to gauge how the quality of scientific software has varied among the community in general we consider the Collected Algorithms from the ACM (CALGO). These codes have been published since 1960 and are widely regarded as being state-of-the-art both in algorithmic and coding terms at the time of publication. In section 4 we report on how our chosen metric values have changed with time by considering the Fortran routines that have appeared in CALGO since 1975.

Finally, in section 5, we draw some conclusions from the results we have presented.

2 Software Metrics

In addition to the number of executable *L*ines *O*f *C*ode (LOC) and the number of explicit GOTO statements in a subprogram, we will also consider three further metrics: cyclomatic complexity [McC76], knot count [WHH79] and a variant of the path count metric proposed by Nejmeh [Nej88].

It should be noted that the metrics used in this paper have been chosen to measure qualities of Fortran code although some of them may also be applied successfully to other procedural languages. Object oriented languages require a different approach and there appears to be, as yet, no general

consensus on which metrics are most appropriate. A general discussion of object oriented software metrics may be found in Lorenz and Kidd [LK94].

Cyclomatic Complexity

The control graph of a program unit is a directed graph whose nodes are the basic blocks of code and whose edges are directed arcs corresponding to the flow of control between the basic blocks. A basic block is a section of code which contains no transfer of control (for example, a sequence of assignment statements). The cyclomatic complexity, $V(G)$, is defined to be the cyclomatic number of the control graph, i.e.,

$$V(G) = \text{Number of edges} - \text{Number of nodes} + 1$$

This may be shown to be equivalent to one more than the number of predicates (decision statements) used in the code and hence, we believe, it is a good indicator of the complexity of the underlying algorithm. This metric was extended by Myers [Mye77] who suggested the use of a cyclomatic complexity interval to take account of the additional complexity caused by compound predicates. The lower bound of Myer's interval is the cyclomatic complexity and the upper bound is defined as one more that the total number of conditions appearing in the code.

Cyclomatic complexity was originally advocated both as a measure of the testing effort required for a module and as an effective way of dividing software into subroutines. Shepperd [She88] and Shepperd and Ince [SI94] have questioned the use of the metric for measuring testing effort as it appears to be extremely insensitive to the structure of the software. They state that this may be due to the fact that the measure is based on a lexical rather than a structural view of the code.

In the present paper the metric is used as an indicator that a piece of code would probably benefit from being broken down into a number of simpler program units. McCabe [McC76] suggested a maximum value of 10 for an individual program unit while Grady [Gra94], after analyzing the relationship between the cyclomatic complexity and the number of updates required to each module in 830,000 lines of Fortran code, suggests a maximum value of 14.

Knot Count

A knot occurs in a piece of code whenever the paths associated with two transfers of control intersect (see Figure 1 for two examples). Code with large knot counts is generally extremely difficult to read and understand. The number of knots is, therefore, a good indicator of code clarity.